HERENCIA

Permite definir una nueva clase (**derivada** o **subclase**) como extensión de una o más clases existentes (**base** o **superclase**). Estas modificaciones generalmente añaden nuevos miembros (variables o funciones), aunque también se puede redefinir las ya existentes. La principal **ventaja** de la herencia es permitir la **reutilización del código**.

- ✓ Si un miembro se redefine en la clase derivada, el nombre redefinido oculta el nombre del elemento de la clase base.
- ✓ Algunos elementos de la clase base no pueden ser heredados:
  - Los constructores y destructores.
  - Las funciones friend.
  - Los datos y funciones static.
  - El operador de asignación (=) sobrecargado.

Modificador **protected**: especifica el control de acceso a los miembros. Permite que los datos y funciones **private** de la *clase base* sean **public** para la *clase derivada*.

Tipos de acceso

En el proceso de herencia la clase base puede ser **public** o **private** para la clase derivada. En el primer caso, se heredan los miembros public y protected con las mismas características. Si la clase base es private todos los datos de la clase derivada serán private.

Tipo dato clase base	clase derivada		clase sin relación con clase base
	clase base **public**	clase base **private**	
private	sin acceso directo	sin acceso directo	sin acceso directo
protected	protected	private	sin acceso directo
public	public	private	accesible mediante operador (**.**) o (→)

Aunque menos utilizado el tipo de herencia también puede ser **protected**, que provoca que los miembros públicos de la clase base sean protected en la clase derivada.

class <Clase_Derivada> **:** {public|**private**|protected} <Clase_Base> // private por defecto
{ /* métodos public, protected y private */ };

Inicializador base

Un objeto de la clase derivada contiene todos los miembros de la clase base que deben ser inicializados. Al definir el constructor de la clase derivada se debe especificar un inicializador base debido a que los constructores no se heredan. El inicializador base es la forma de llamar a los constructores de las clases base para inicializar las variables miembro heredadas. Se especifica poniendo a continuación de los argumentos del constructor de la clase derivada el carácter '**:**' y el nombre del constructor de la(s) clase(s) base, seguido de la lista de argumentos.

```
<Clase_Derivada>(argumentos para la clase derivada) :
        <Clase_Base>(argumentos para la clase base)          // constructores
{  /* funciones de inicialización  */  };
```

- ✓ El inicializador base puede ser omitido en el caso de que la clase base tenga un constructor por defecto.
- ✓ En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

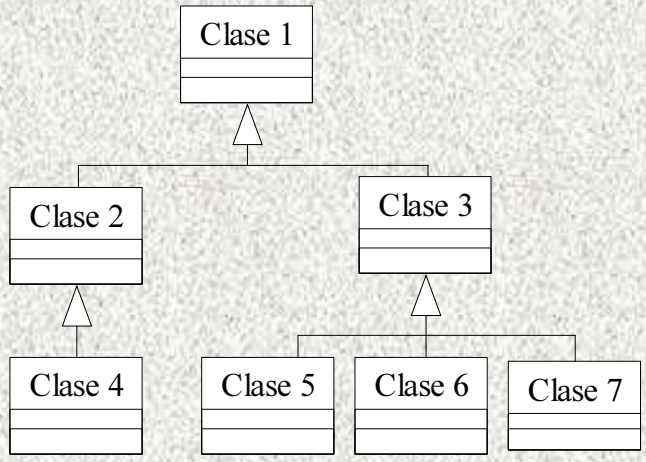
Inicializador base – ejemplo

```
class base
{
   public:
      int num;
      base(void) { num = 1; cout << num << endl; }
      base(int val) { num = val; cout << num << endl; }
};

class derivada : public base
{
   public:
      int num;
      derivada(void) { num = 10; cout << num << endl; }
      derivada(int valor) : base(valor/10) { num = valor; cout << num << endl; }
};

int main()
{
   derivada obj1,
            obj2 = 20;
   return 0;
}
```

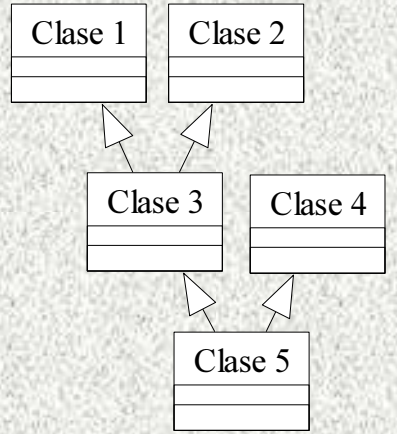
Salida:

Herencia simple y múltiple

Una clase puede heredar variables y funciones miembro de más de una clase base. En el caso de que herede de una única clase es una herencia simple y en el caso de que herede de varias clases base es un caso de herencia múltiple.



Herencia simple



Herencia múltiple

class <C_Derivada> > **:** // lista de clases base

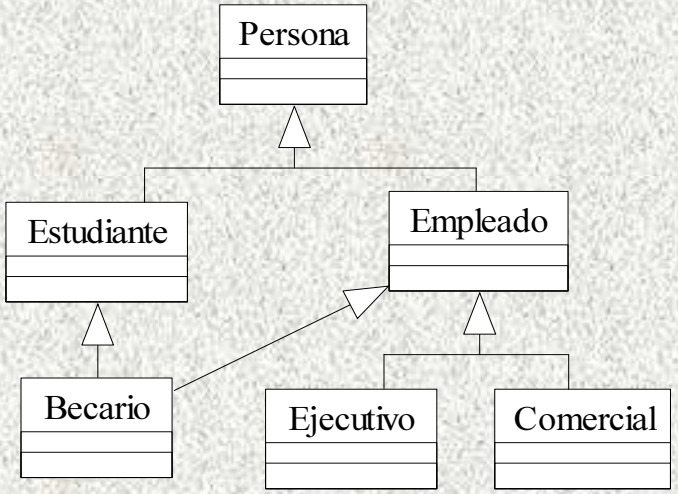
{public|private} <C_Base1> [, {public|private} C_Base2 [, ...]]

{ /* métodos public, protected y private */ }; // declaraciones clase derivada

Ej. 01: H. Simple

Jerarquía de clases

La clase derivada puede a su vez puede ser clase base en un nuevo proceso de derivación, iniciando una **jerarquía de clases**, que puede ser tan compleja como sea necesario. Se usa especialmente en la resolución de problemas complejos; no para resolver los sencillos.



- ✓ Cada vez que se crea un objeto de un tipo derivado, se crea un sólo objeto que reúne los atributos de esa clase y los de las clases bases (al crear un objeto *Comercial*, también se crea un *Empleado* y una *Persona*, por lo que tendrá los atributos de las tres clases).
- ✓ Siempre se pueden crear nuevas clases para resolver nuevos problemas (*Becario*).
- ✓ Se pueden aplicar procedimientos genéricos a una clase que afectarán a todos los heredados sin tener que compilar nuevamente todos los archivos fuente involucrados.
- ✓ Se debe estudiar muy bien el diseño de las clases bases, porque pueden presentarse más inconvenientes que ventajas al utilizarlas.

Conversiones entre objetos

Es posible realizar conversiones o asignaciones de un objeto de una clase derivada a un objeto de la clase base (de lo particular a lo general). Se puede perder información, pues existirán variables de la clase derivada que no tengan a qué asignarse.

Objeto_clase_base = Objeto_clase_derivada // algunas variables no existen en clase_base

Las conversiones o asignaciones de lo general a lo particular no son posibles, porque puede suceder que no se dispongan de valores para todas las variables miembro de la clase derivada.

Objeto_clase_derivada = Objeto_clase_base // incorrecto, quedan variables sin valor

Redefinición de funciones

En una clase derivada se puede definir una función que ya exista en la clase base (*overriding* o superposición). La definición en la clase derivada oculta **todas** las funciones con ese mismo nombre en la clase base aunque difieran en los valores de retorno o tengan distinto número o tipo de parámetros (función sobrecargada). Sólo son accesibles utilizando el nombre completo.

objeto_clase_derivada.clase_base :: método(parámetros);

Constructores y destructores en clases compuestas

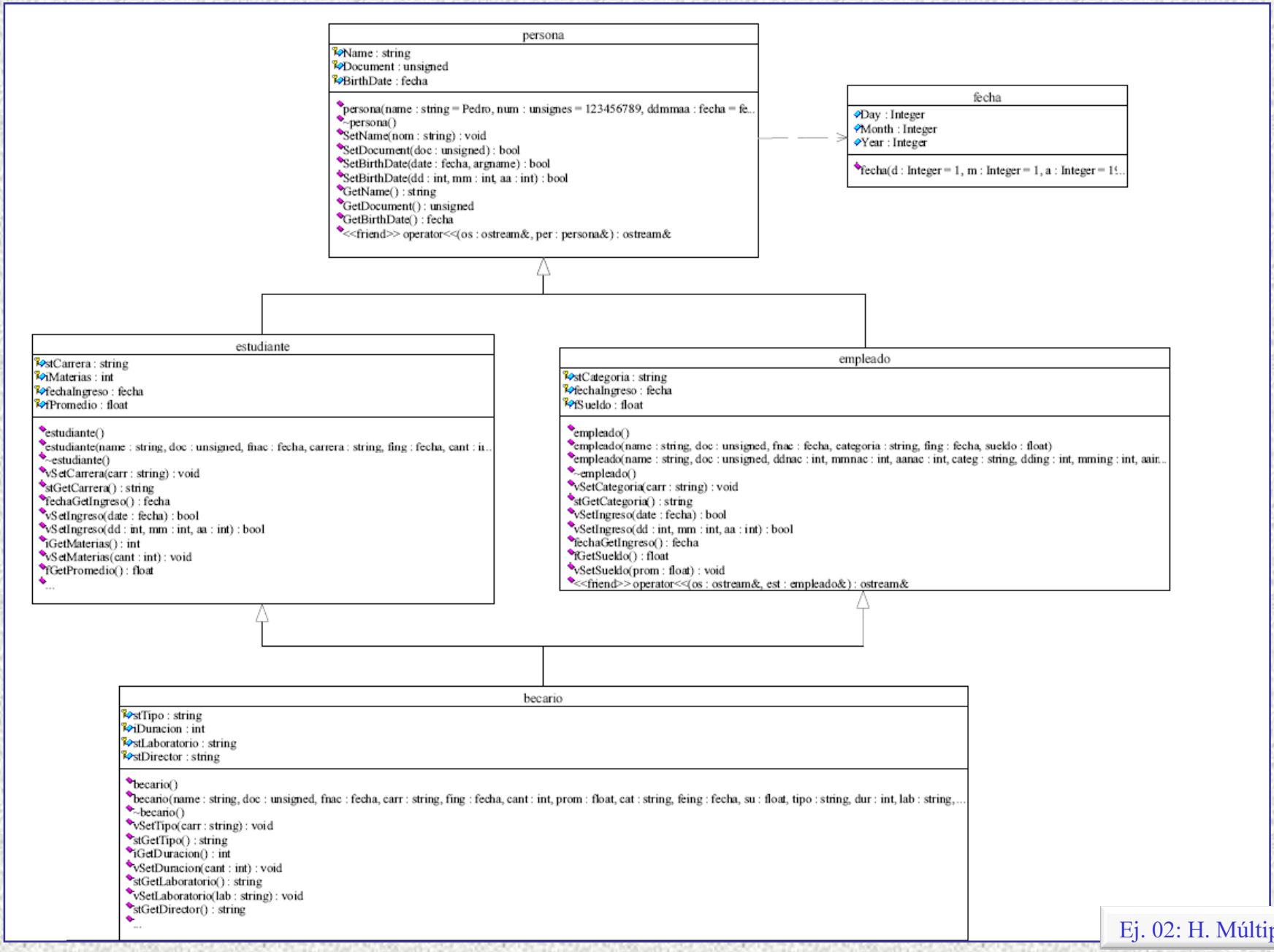
Primero se ejecutan los constructores de las clases base, luego el código el constructor de los objetos miembro y finalmente el de la clase derivada. El destructor sigue el orden inverso.

```
class base
{
   public:
      base(void) {cout<<"Constructor base"<< endl;}
      ~base(void) {cout<<"Destructor base"<<endl;}
};

class derivada : public base
{
   public:
       miembro obj;
      derivada(void) {cout<<"Constructor derivada\n"; }
      ~derivada(void) {cout<<"Destructor derivada\n"; }
};

int main()
{
   derivada obj;
   return 0;
}
```

```
class miembro
{
   public:
      miembro(void) {cout<<"Constructor  miembro \n";}
      ~miembro(void) {cout <<"Destructor  miembro\n";}
};
```

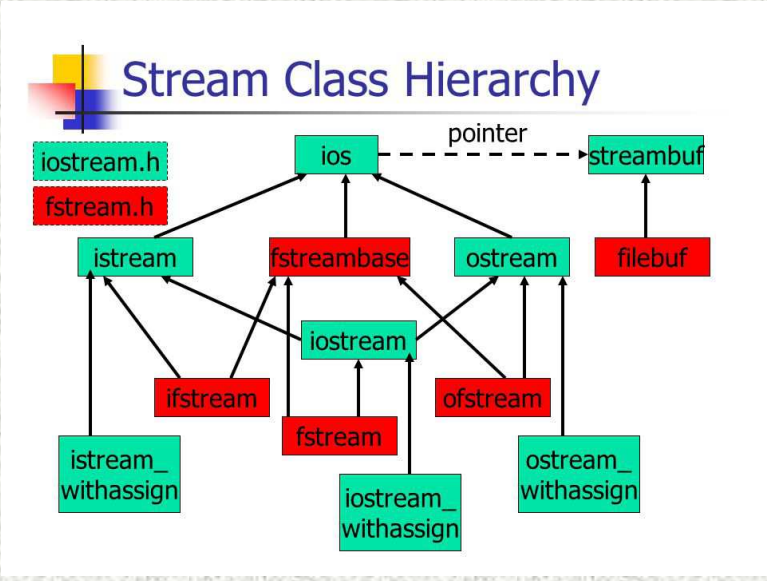
Salida:
```
Constructor base
Constructor  miembro
Constructor derivada
Destructor derivada
Destructor  miembro
Destructor base
```

Ejemplo herencia múltiple: clase becario



Manejo de archivos

Para poder leer desde o escribir en ficheros se debe incluir la librería ***fstream*** (sin .h). En ella se definen las clases ***ifstream*** (in), ***ofstream*** (out) y ***fstream*** (in / out), derivadas de *istream* y *ostream*, que a su vez derivan de la clase *ios*, que tiene un puntero a un objeto ***streambuf***.



Antes de *abrir un fichero* hay que crear un objeto de alguna de las clases e indicar el modo de apertura:

- ✓ **ios::in**: para leer datos.
- ✓ **ios::out**: para escribir datos
- ✓ **ios::app**: para añadir los datos al final del fichero (sino el puntero de lectura puede haber fijado la posición de la próxima escritura (usando in|out) y perder información).
- ✓ **ios::ate**: abre el fichero y sitúa el cursor al final del mismo (cuidado al leer!).
- ✓ **ios::trunc**: sobrescribe el fichero si existe (fija el tamaño a 0 bytes), sino lo crea.
- ✓ **ios::binary**: fija el acceso en modo binario en vez de texto.

Los flags de modo se pueden combinar con el operador bitwaise OR (|).

Manejo de archivos

```
/***** Ejemplo:

    fstream  fichero_in,
             fichero_out,
             fichero_in_out;

    fichero_in.open("datos_entrada.dat", ios::in);

    fichero_out.open("datos_salida.dat", ios::out);

    fichero_in_out.open(“datos.dat”, ios::app|ios::in|ios::out);

fin del ejemplo *****/
```

- Las clases tienen también constructores que permiten abrir ficheros de forma automática:

  ifstream fichero("datos.dat"); // modo lectura por utilizar *ifsteam*

- Como se trabaja con streams, las operaciones de entrada salida son las mismas para el caso de teclado/pantalla que para los ficheros. Las funciones miembro más utilizadas son: put(*char*), write(*const char**, *int*), get(en sus tres formatos), getline(*char**, *int*, *char* c=‘\n’), read(*char**, *int*) y flush(). Todas retornar una referencia a un objeto ostream ó istream.

Manejo de archivos – *fstream*

- **fstream**(): constructor por defecto. Construye un flujo sin abrir ningún fichero.
- **fstream**(*const char** fname, *int* mode, *int* prot = filebuf::openprot): constructor general que crea un flujo al fichero, en el modo indicado (ios::in|out|append), y con la protección que por defecto equivale a sh_compat.
  - ✓ filebuf::sh_compat: modo compatible (MS-DOS).
  - ✓ filebuf::sh_none: modo exclusivo, no se comparte.
  - ✓ filebuf:: sh_read: se permite compartir para lectura.
  - ✓ filebuf::sh_ write: se permite compartir para escritura.
  - ✓ filebuf::sh_none: modo exclusivo, no se comparte.
  - ✓ ios::append: para añadir datos al final del fichero
- *void* **open**(*const char** fname, *int* mode, *int* prot = filebuf::openprot): abre un fichero.
- *int* **is_open**(): devuelve un valor no nulo si el fichero está abierto.
- *void* **close**(): cierra el fichero asociado con un flujo sin destruirlo.
- operadores >> y <<: se pueden utilizar como para entrada/salida en el modo texto.

Ej. 03: Arch texto

Ficheros binarios

Los archivos binarios son útiles para almacenar estructuras completas, en las que se mezclan strings con datos numéricos, con un mínimo de espacio (en archivos de texto se puede guardar la misma información pero en mayor tamaño). Para la entrada y salida de datos se utilizan los métodos read y write.

```
/***** Ejemplo:

#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct persona  {   char nombre[30];    int edad;   float altura;  };

int main()
{
   persona Pedro, Pablo;

   strcpy(Pedro.nombre, "Pedro Picapiedra");
   Pedro.edad = 59;    Pedro.altura = 1.80;

   ofstream fsalida("datos.dat", ios::out|ios::binary);
   fsalida.write(reinterpret_cast<char *>(&Pedro), sizeof(persona));
   fsalida.close();

   ifstream fentrada("datos.dat", ios::in|ios::binary);
   fentrada.read(reinterpret_cast<char*>(&Pablo), sizeof(persona));
   fentrada.close();

    cout << Pablo.nombre << ", " << Pablo.edad << " anios "
         << Pablo.altura << "m" << endl;

    return 0;
}

fin del ejemplo *****/
```

Ej. 04: Arch binario

Ficheros de acceso aleatorio

El acceso no se realiza de modo secuencial, sino que pueden hacerse lecturas o escrituras en cualquier punto del archivo. En la librería ios se dispone de un tipo enumerado para indicar movimientos relativos dentro de un stream: *enum seek_dir* = {beg, cur, end}.

- *istream&* **seekg**(*streampos* pos): cambia la posición del cursor en streams de entrada en forma absoluta.
- *istream&* **seekg**(*streamoff* offset, *seek_dir* dir): realiza cambios relativos desde el principio del archivo, desde la posición actual o desde el final (dir = ios::beg|cur|end).
- *ostream&* **seekp**(…):equivalente a las versiones de seekg aplicado al stream de salida.
- *streampos* ***tellg***(): devuelve la posición actual del cursor dentro de un stream de entrada.
- *streampos* ***tellp***(): devuelve la posición actual del cursor dentro de un stream de salida.

La resolución de las funciones seek es de un byte. Cuando se diseñen streams propios para nuevas clases, derivándolas de ifstream, ofstream o fstream, sería conveniente sobrecargar las funciones seek y tell para que trabajen a nivel de registro, en lugar de hacerlo a nivel de byte.

Errores de Entrada/Salida

La clase *ios* define una variable *enum* llamada **io_state** con los siguientes valores: goodbit, eofbit, badbit y failbit. Cada flujo mantiene información sobre los errores que se hayan podido producir. Esta información se puede chequear con las siguientes funciones:

- *int* ***good***(): devuelve un valor distinto de cero (true) si no ha habido errores.
- *int* ***eof***(): devuelve un valor distinto de cero si se ha llegado al fin del fichero.
- *int* ***bad***(): devuelve un valor distinto de cero si ha producido un error grave de E/S. No se puede continuar en esas condiciones.
- *int* ***fail***(): devuelve un valor distinto de cero si se ha producido cualquier error de E/S distinto de EOF. Si una llamada a *bad*() devuelve 0 (no error de ese tipo), el error puede no ser grave y la lectura puede proseguir después de llamar a la función *clear*().
- *int* ***clear***(): se borran los bits de error que puedan estar activados.

Tanto los operadores sobrecargados (<< y >>), como las funciones miembro de E/S devuelven referencias al flujo correspondiente para poder utilizar *if* o *while* para saber si se ha producido un error o una condición de fin de fichero.

Ej. 05 Arch pos

Clase *sstream*

Permite trabajar con streams strings. La librería es similar a iostream y fstream con la diferencia que las operaciones de E/S se realizan sobre strings. Provee funcionalidad equivalente a las funciones sscanf() y sprintf() de la librería estándar de C. Incluye tres clases:

- ✓ **sstringtream**: operaciones de entrada y salida.
- ✓ **istringtream**: sólo operaciones de entrada.
- ✓ **ostringtream**: sólo operaciones de salida.

Dado que son subclases de iostreams, se encuentran disponibles las funciones anteriores. Cuenta también con los operadores << y >> sobrecargados para realizar conversiones entre strings y tipos numéricos.

```
/***** Ejemplo:

#include <sstream>
#include <iostream>

using namespace std;

int main()
{
   int numi = 3;
   float numf = 2.7182;
   stringstream ss1, ss2;

   ss1 << numi;
   ss2 << numf;
   cout << ss1.str() << endl << ss2.str() << endl;

   ss1.str("1959");
   ss1 >> numi;
   ss2.str("-2.18e-03");
   ss2 >> numf;
   cout << numi << endl << numf << endl;

   return 0;
}

fin del ejemplo *****/
```